Chapter 1

Java Building Blocks

OCA EXAM OBJECTIVES COVERED IN THIS CHAPTER:

✓ **Java Basics**

- Define the scope of variables
- Define the structure of a Java class
- Create executable Java applications with a main method; run a Java program from the command line; including console output
- Import other Java packages to make them accessible in your code
- Compare and contrast the features and components of Java such as platform independence, object orientation, encapsulation, etc.

✓ **Working with Java Data Types**

- Declare and initialize variables (including casting or primitive types)
- Differentiate between object reference variables and primitive variables
- Know how to read or write to object fields
- Explain an Object's Lifecycle (creation, "dereference by reassignment" and garbage collection

Welcome to the beginning of your journey to become certified on Java. We assume this isn't the first Java programming book you've read. Although we do talk about the basics, we do so only because we want to make sure you have all the terminology and detail you'll need for the OCA exam. If you've never written a Java program before, we recommend you pick up an introductory book on any version of Java—something like *Head First Java, 2nd Edition* (O'Reilly Media, 2005); *Java for Dummies* (For Dummies, 2014), or *Thinking in Java, 4th Edition* (Prentice Hall, 2006). (It's okay if the book covers an older version of Java—even Java 1.3 is fine.) Then come back to this certification study guide.

This chapter covers the fundamentals of Java. You'll see how to define and run a Java class, and learn about packages, variables, and the object life cycle.

Understanding the Java Class Structure

In Java programs, classes are the basic building blocks. When defining a *class*, you describe all the parts and characteristics of one of those building blocks. To use most classes, you have to create objects. An *object* is a runtime instance of a class in memory. All the various objects of all the different classes represent the state of your program.

In the following sections, we'll look at fields, methods, and comments. We'll also explore the relationship between classes and files.

Fields and Methods

Java classes have two primary elements: *methods*, often called functions or procedures in other languages, and *fields*, more generally known as variables. Together these are called the *members* of the class. Variables hold the state of the program, and methods operate on that state. If the change is important to remember, a variable stores that change. That's all classes really do. It's the programmer who creates and arranges these elements in such a way that the resulting code is useful and, ideally, easy for other programmers to understand.

Other building blocks include interfaces, which you'll learn about in Chapter 5, "Class Design," and enums, which you'll learn about when you start studying for the OCP exam.

The simplest Java class you can write looks like this:

```
1: public class Animal {
2: }
```

Java calls a word with special meaning a *keyword*. The `public` keyword on line 1 means the class can be used by other classes. The `class` keyword indicates you're defining a class. `Animal` gives the name of the class. Granted, this isn't a very interesting class, so add your first field:

```
1: public class Animal {
2:  String name;
3: }
```

The line numbers aren't part of the program; they're just there to make the code easier to talk about.

On line 2, we define a variable named `name`. We also define the type of that variable to be a `String`. A `String` is a value that we can put into, such as **`"this is a string"`**. `String` is also a class supplied with Java. Next you can add methods:

```
1: public class Animal {
2:  String name;
3:  public String getName() {
4:    return name;
5:  }
6:  public void setName(String newName) {
7:    name = newName;
8:  }
9: }
```

On lines 3–5, you've defined your first method. A method is an operation that can be called. Again, `public` is used to signify that this method may be called from other classes. Next comes the return type—in this case, the method returns a `String`. On lines 6–8 is another method. This one has a special return type called *void*. `void` means that no value at all is returned. This method requires information be supplied to it from the calling method; this information is called a *parameter*. `setName` has one parameter named *newName*, and it is of type `String`. This means the caller should pass in one `String` parameter and expect nothing to be returned.

The full declaration of a method is called a *method signature*. In this example, can you identify the return type and parameters?

```
public int numberVisitors(int month)
```

The return type is `int`, which is a numeric type. There's one parameter named *`month`*, which is of type `int` as well.

Comments

Another common part of the code is called a *comment*. Because comments aren't executable code, you can place them anywhere. Comments make your code easier to read. You won't see many comments on the exam—the exam creators are trying to make the code difficult to read—but you'll see them in this book as we explain the code. And we hope you use them in your own code. There are three types of comments in Java. The first is called a single-line comment:

```
// comment until end of line
```

A single-line comment begins with two slashes. Anything you type after that on the same line is ignored by the compiler. Next comes the multiple-line comment:

```
/* Multiple
 * line comment
 */
```

A multiple-line comment (also known as a multiline comment) includes anything starting from the symbol `/*` until the symbol `*/`. People often type an asterisk (`*`) at the beginning of each line of a multiline comment to make it easier to read, but you don't have to. Finally, we have a Javadoc comment:

```
/**
 * Javadoc multiple-line comment
 * @author Jeanne and Scott
 */
```

This comment is similar to a multiline comment except it starts with `/**`. This special syntax tells the Javadoc tool to pay attention to the comment. Javadoc comments have a specific structure that the Javadoc tool knows how to read. You won't see a Javadoc comment on the exam—just remember it exists so you can read up on it online when you start writing programs for others to use.

As a bit of practice, can you identify which type of comment each of these five words is in? Is it a single-line or a multiline comment?

```
/*
 * // anteater
 */
// bear
// // cat
```

```
// /* dog */
/* elephant */
/*
 * /* ferret */
 */
```

Did you look closely? Some of these are tricky. Even though comments technically aren't on the exam, it is good to practice to look at code carefully.

Okay, on to the answers. `anteater` is in a multiline comment. Everything between `/*` and `*/` is part of a multiline comment—even if it includes a single-line comment within it! `bear` is your basic single-line comment. `cat` and `dog` are also single-line comments. Everything from `//` to the end of the line is part of the comment, even if it is another type of comment. `elephant` is your basic multiline comment.

The line with `ferret` is interesting in that it doesn't compile. Everything from the first `/*` to the first `*/` is part of the comment, which means the compiler sees something like this:

```
/* */ */
```

We have a problem. There is an extra `*/`. That's not valid syntax—a fact the compiler is happy to inform you about.

Classes vs. Files

Most of the time, each Java class is defined in its own `*.java` file. It is usually `public`, which means any code can call it. Interestingly, Java does not require that the class be `public`. For example, this class is just fine:

```
1: class Animal {
2:  String name;
3: }
```

You can even put two classes in the same file. When you do so, at most one of the classes in the file is allowed to be public. That means a file containing the following is also fine:

```
1: public class Animal {
2:   private String name;
3: }
4: class Animal2 {
5: }
```

If you do have a public class, it needs to match the filename. `public class Animal2` would not compile in a file named `Animal.java`. In Chapter 5, we will discuss what non-public access means.

Writing a *main()* Method

A Java program begins execution with its `main()` *method*. A `main()` method is the gateway between the startup of a Java process, which is managed by the *Java Virtual Machine* (JVM), and the beginning of the programmer's code. The JVM calls on the underlying system to allocate memory and CPU time, access files, and so on.

The `main()` method lets us hook our code into this process, keeping it alive long enough to do the work we've coded. The simplest possible class with a `main()` method looks like this:

```
1: public class Zoo {
2:  public static void main(String[] args) {
3:
4:  }
5:}
```

This code doesn't do anything useful (or harmful). It has no instructions other than to declare the entry point. It does illustrate, in a sense, that what you can put in a `main()` method is arbitrary. Any legal Java code will do. In fact, the only reason we even need a class structure to start a Java program is because the language requires it. To compile and execute this code, type it into a file called `Zoo.java` and execute the following:

```
$ javac Zoo.java
$ java Zoo
```

If you don't get any error messages, you were successful. If you do get error messages, check that you've installed a Java Development Kit (JDK) and not a Java Runtime Environment (JRE), that you have added it to the `PATH`, and that you didn't make any typos in the example. If you have any of these problems and don't know what to do, post a question with the error message you received in the Beginning Java forum at CodeRanch (`www.coderanch.com/forums/f-33/java`).

To compile Java code, the file must have the extension `.java`. The name of the file must match the name of the class. The result is a file of *bytecode* by the same name, but with a `.class` filename extension. Bytecode consists of instructions that the JVM knows how to execute. Notice that we must omit the `.class` extension to run `Zoo.java` because the period has a reserved meaning in the JVM.

The rules for what a Java code file contains, and in what order, are more detailed than what we have explained so far (there is more on this topic later in the chapter). To keep things simple for now, we'll follow a subset of the rules:

- Each file can contain only one class.
- The filename must match the class name, including case, and have a `.java` extension.

Suppose we replace line 3 in `Zoo.java` with `System.out.println("Welcome!");`. When we compile and run the code again, we'll get the line of output that matches what's between the quotes. In other words, the program will output `Welcome!`.

Let's first review the words in the `main()` method's signature, one at a time. The keyword `public` is what's called an *access modifier*. It declares this method's level of exposure to potential callers in the program. Naturally, `public` means anyplace in the program. You'll learn about access modifiers in Chapter 4, "Methods and Encapsulation."

The keyword *static* binds a method to its class so it can be called by just the class name, as in, for example, `Zoo.main()`. Java doesn't need to create an object to call the `main()` method—which is good since you haven't learned about creating objects yet! In fact, the JVM does this, more or less, when loading the class name given to it. If a `main()` method isn't present in the class we name with the `.java` executable, the process will throw an error and terminate. Even if a `main()` method is present, Java will throw an exception if it isn't static. A nonstatic `main()` method might as well be invisible from the point of view of the JVM. We'll see `static` again in Chapter 4.

The keyword *void* represents the *return type*. A method that returns no data returns control to the caller silently. In general, it's good practice to use `void` for methods that change an object's state. In that sense, the `main()` method changes the program state from started to finished. We will explore return types in Chapter 4 as well. Excited for Chapter 4 yet?

Finally we arrive at the `main()` method's parameter list, represented as an array of `java.lang.String` objects. In practice, you can write `String[] args`, `String args[]` or `String... args`; the compiler accepts any of these. The variable name *args* hints that this list contains values that were read in (arguments) when the JVM started. You can use any name you like, though. The characters `[]` are brackets and represent an array. An array is a fixed-size list of items that are all of the same type. The characters `...` are called varargs (variable argument lists). You will learn about `String` in Chapter 2, "Operators and Statements." Arrays and varargs will follow in Chapter 3, "Core Java APIs."

Let's see how to use the *args* parameter. First we modify the Zoo program to print out the first two arguments passed in:

```
public class Zoo {
  public static void main(String[] args) {
    System.out.println(args[0]);
   System.out.println(args[1]);
} }
```

`args[0]` accesses the first element of the array. That's right: array indexes begin with 0 in Java. To run it, type this:

```
$ javac Zoo.java
$ java Zoo Bronx Zoo
```

The output is what you might expect:

```
Bronx
Zoo
```

The program correctly identifies the first two "words" as the arguments. Spaces are used to separate the arguments. If you want spaces inside an argument, you need to use quotes as in this example:

```
$ javac Zoo.java
$ java Zoo "San Diego" Zoo
```

Now we have a space in the output:

```
San Diego
Zoo
```

All command-line arguments are treated as `String` objects, even if they represent another data type:

```
$ javac Zoo.java
$ java Zoo Zoo 2
```

No matter. You still get the values output as `String`s. In Chapter 2, you'll learn how to convert `String`s to numbers.

```
Zoo
2
```

Finally, what happens if you don't pass in enough arguments?

```
$ javac Zoo.java
$ java Zoo Zoo
```

Reading `args[0]` goes fine and `Zoo` is printed out. Then Java panics. There's no second argument! What to do? Java prints out an exception telling you it has no idea what to do with this argument at position 1. (You'll learn about exceptions in Chapter 6, "Exceptions.")

```
        ZooException in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 1
       at mainmethod.Zoo.main(Zoo.java:7)
```

To review, you need to have a JDK to compile because it includes a compiler. You do not need to have a JDK to run the code—a JRE is enough. Java class files run on the JVM and therefore run on any machine with Java rather than just the machine or operating system they happened to have been compiled on.

Understanding Package Declarations and Imports

Java comes with thousands of built-in classes, and there are countless more from developers like you. With all those classes, Java needs a way to organize them. It handles this in a way similar to a file cabinet. You put all your pieces of paper in folders. Java puts classes in *packages*. These are logical groupings for classes.

We wouldn't put you in front of a file cabinet and tell you to find a specific paper. Instead, we'd tell you which folder to look in. Java works the same way. It needs you to tell it which packages to look in to find code.

Suppose you try to compile this code:

```
public class ImportExample {
  public static void main(String[] args) {
    Random r = new Random();   // DOES NOT COMPILE
    System.out.println(r.nextInt(10));
  }
}
```

The Java compiler helpfully gives you an error that looks like this:

```
Random cannot be resolved to a type
```

This error could mean you made a typo in the name of the class. You double-check and discover that you didn't. The other cause of this error is omitting a needed *import* statement. Import statements tell Java which packages to look in for classes. Since you didn't tell Java where to look for `Random`, it has no clue.

Trying this again with the import allows you to compile:

```
import java.util.Random;  // import tells us where to find Random
public class ImportExample {
  public static void main(String[] args) {
    Random r = new Random();
    System.out.println(r.nextInt(10));  // print a number between 0 and 9
   }
}
```

Now the code runs; it prints out a random number between 0 and 9. Just like arrays, Java likes to begin counting with 0.

Java classes are grouped into packages. The import statement tells the compiler which package to look in to find a class. This is similar to how mailing a letter works.

Imagine you are mailing a letter to 123 Main St., Apartment 9. The mail carrier first brings the letter to 123 Main St. Then she looks for the mailbox for apartment number 9. The address is like the package name in Java. The apartment number is like the class name in Java. Just as the mail carrier only looks at apartment numbers in the building, Java only looks for class names in the package.

Package names are hierarchical like the mail as well. The postal service starts with the top level, looking at your country first. You start reading a package name at the beginning too. If it begins with `java` or `javax`, this means it came with the JDK. If it starts with something else, it likely shows where it came from using the website name in reverse. From example, `com.amazon.java8book` tells us the code came from amazon.com. After the website name, you can add whatever you want. For example, `com.amazon.java8.my.name` also came from amazon.com. Java calls more detailed packages *child packages*. `com.amazon.java8book` is a child package of `com.amazon`. You can tell because it's longer and thus more specific.

You'll see package names on the exam that don't follow this convention. Don't be surprised to see package names like `a.b.c`. The rule for package names is that they are mostly letters or numbers separated by dots. Technically, you're allowed a couple of other characters between the dots. The rules are the same as for variable names, which you'll see later in the chapter. The exam may try to trick you with invalid variable names. Luckily, it doesn't try to trick you by giving invalid package names.

In the following sections, we'll look at imports with wildcards, naming conflicts with imports, how to create a package of your own, and how the exam formats code.

Wildcards

Classes in the same package are often imported together. You can use a shortcut to import all the classes in a package:

```
import java.util.*;    // imports java.util.Random among other things
public class ImportExample {
  public static void main(String[] args) {
    Random r = new Random();
    System.out.println(r.nextInt(10));
  }
}
```

In this example, we imported `java.util.Random` and a pile of other classes. The `*` is a wildcard that matches all classes in the package. Every class in the `java.util` package is available to this program when Java compiles it. It doesn't import child packages, fields, or methods; it imports only classes. (Okay, it's only classes for now, but there's a special type of import called the "static import" that imports other types. You'll learn more about that in Chapter 4.)

You might think that including so many classes slows down your program, but it doesn't. The compiler figures out what's actually needed. Which approach you choose is personal preference.

Listing the classes used makes the code easier to read, especially for new programmers. Using the wildcard can shorten the import list. You'll see both approaches on the exam.

Redundant Imports

Wait a minute! We've been referring to `System` without an import and Java found it just fine. There's one special package in the Java world called `java.lang`. This package is special in that it is automatically imported. You can still type this package in an `import` statement, but you don't have to. In the following code, how many of the imports do you think are redundant?

```
1: import java.lang.System;
2: import java.lang.*;
3: import java.util.Random;
4: import java.util.*;
5: public class ImportExample {
6:  public static void main(String[] args) {
7:    Random r = new Random();
8:    System.out.println(r.nextInt(10));
9:  }
10: }
```

The answer is that three of the imports are redundant. Lines 1 and 2 are redundant because everything in `java.lang` is automatically considered to be imported. Line 4 is also redundant in this example because `Random` is already imported from `java.util.Random`. If line 3 wasn't present, `java.util.*` wouldn't be redundant, though, since it would cover importing `Random`.

Another case of redundancy involves importing a class that is in the same package as the class importing it. Java automatically looks in the current package for other classes.

Let's take a look at one more example to make sure you understand the edge cases for imports. For this example, `Files` and `Paths` are both in the package `java.nio.file`. You don't need to memorize this package for the OCA exam (but you should know it for the OCP exam). When testing your understanding of packages and imports, the OCA exam will use packages you may never have seen before. The question will let you know which package the class is in if you need to know that in order to answer the question.

What imports do you think would work to get this code to compile?

```
public class InputImports {
  public void read(Files files) {
    Paths.get("name");
  }
}
```

There are two possible answers. The shorter one is to use a wildcard to import both at the same time:

```
import java.nio.file.*;
```

The other answer is to import both classes explicitly:

```
import java.nio.file.Files;
import java.nio.file.Paths;
```

Now let's consider some imports that don't work:

```
import java.nio.*; // NO GOOD - a wildcard only matches
                   //class names, not "file.*Files"
import java.nio.*.*; // NO GOOD - you can only have one wildcard
                     //and it must be at the end
import java.nio.files.Paths.*; // NO GOOD - you cannot import methods
                               //only class names
```

Naming Conflicts

One of the reasons for using packages is so that class names don't have to be unique across all of Java. This means you'll sometimes want to import a class that can be found in multiple places. A common example of this is the `Date` class. Java provides implementations of `java.util.Date` and `java.sql.Date`. This is another example where you don't need to know the package names for the OCA exam—they will be provided to you. What import could we use if we want the `java.util.Date` version?

```
public class Conflicts {
  Date date;
  // some more code
}
```

The answer should be easy by now. You can write either **`import java.util.*;`** or **`import java.util.Date;`**. The tricky cases come about when other imports are present:

```
import java.util.*;
import java.sql.*; // DOES NOT COMPILE
```

When the class is found in multiple packages, Java gives you the compiler error:

```
The type Date is ambiguous
```

In our example, the solution is easy—remove the `java.sql.Date` import that we don't need. But what do we do if we need a whole pile of other classes in the `java.sql` package?

```
import java.util.Date;
import java.sql.*;
```

Ah, now it works. If you explicitly import a class name, it takes precedence over any wildcards present. Java thinks, "Okay! The programmer really wants me to assume use of the `java.util.Date` class."

One more example. What does Java do with "ties" for precedence?

```
import java.util.Date;
import java.sql.Date;
```

Java is smart enough to detect that this code is no good. As a programmer, you've claimed to explicitly want the default to be both the `java.util.Date` and `java.sql.Date` implementations. Because there can't be two defaults, the compiler tells you:

```
The import java.sql.Date collides with another import statement
```

If You Really Need to Use Two Classes with the Same Name...

Sometimes you really do want to use `Date` from two different packages. When this happens, you can pick one to use in the import and use the other's fully qualified class name (the package name, a dot, and the class name) to specify that it's special. For example:

```
import java.util.Date;

public class Conflicts {
        Date date;
        java.sql.Date sqlDate;

}
```

Or you could have neither with an import and always use the fully qualified class name:

```
public class Conflicts {
        java.util.Date date;
        java.sql.Date sqlDate;

}
```

Creating a New Package

Up to now, all the code we've written in this chapter has been in the *default package*. This is a special unnamed package that you should use only for throwaway code. You can tell the code is in the default package, because there's no package name. On the exam, you'll see the default package used a lot to save space in code listings. In real life, always name your packages to avoid naming conflicts and to allow others to reuse your code.

Now it's time to create a new package. The directory structure on your computer is related to the package name. Suppose we have these two classes:

```
C:\temp\packagea\ClassA.java
```

```
package packagea;
public class ClassA {
}
```

```
C:\temp\packageb\ClassB.java
```

```
package packageb;
import packagea.ClassA;
public class ClassB {
  public static void main(String[] args) {
    ClassA a;
    System.out.println("Got it");
  }
}
```

When you run a Java program, Java knows where to look for those package names. In this case, running from `C:\temp` works because both `packagea` and `packageb` are underneath it.

Compiling Code with Packages

You'll learn Java much more easily by using the command line to compile and test your examples. Once you know the Java syntax well, you can switch to an integrated development environment (IDE) like Eclipse. An IDE will save you time in coding. But for the exam, your goal is to know details about the language and not have the IDE hide them for you.

Follow this example to make sure you know how to use the command line. If you have any problems following this procedure, post a question in the Beginning Java forum at CodeRanch (`www.coderanch.com/forums/f-33/java`). Describe what you tried and what the error said.

Windows Setup

Create the two files:

- `C:\temp\packagea\ClassA.java`
- `C:\temp\packageb\ClassB.java`

Then type this command:

```
cd C:\temp
```

Mac/Linux Setup

Create the two files:

- `/tmp/packagea/ClassA.java`
- `/tmp/packageb/ClassB.java`

Then type this command:

```
cd /tmp
```

To Compile

Type this command:

```
javac packagea/ClassA.java packageb/ClassB.java
```

If this command doesn't work, you'll get an error message. Check your files carefully for typos against the provided files. If the command does work, two new files will be created:

`packagea/ClassA.class` and `packageb/ClassB.class`.

To Run

Type this command:

```
java packageb.ClassB
```

If it works, you'll see `Got it` printed. You might have noticed we typed `ClassB` rather than `ClassB.class`. In Java you don't pass the extension when running a program.

Class Paths and JARs

You can also specify the location of the other files explicitly using a class path. This technique is useful when the class files are located elsewhere or in special JAR files. A JAR file is like a zip file of mainly Java class files. This goes beyond what you'll need to do on version 8 of the exam, although it appears on older versions.

On Windows, you type the following:

```
java -cp ".;C:\temp\someOtherLocation;c:\temp\myJar.jar" myPackage.MyClass
```

And on Mac OS/Linux, you type this:

```
java -cp ".:/tmp/someOtherLocation:/tmp/myJar.jar" myPackage.MyClass
```

The dot indicates you want to include the current directory in the class path. The rest of the command says to look for loose class files (or packages) in `someOtherLocation` and within `myJar.jar`. Windows uses semicolons to separate parts of the class path; other operating systems use colons.

Finally, you can use a wildcard (*) to match all the JARs in a directory. Here's an example:

```
java -cp "C:\temp\directoryWithJars\*" myPackage.MyClass
```

This command will add all the JARs to the class path that are in `directoryWithJars`. It won't include any JARs in the class path that are in a subdirectory of `directoryWithJars`.

Code Formatting on the Exam

Not all questions will include the imports. If the exam isn't asking about imports in the question, it will often omit the imports to save space. You'll see examples with line numbers that don't begin with 1 in this case. The question is telling you, "Don't worry—imagine the code we omitted is correct; just focus on what I'm giving you." This means when you do see the line number 1 or no line numbers at all, you have to make sure imports aren't missing. Another thing the exam does to save space is to merge code on the same line. You should expect to see code like the following and to be asked whether it compiles. (You'll learn about `ArrayList` in Chapter 3—assume that part is good for now.)

```
6: public void method(ArrayList list) {
7:  if (list.isEmpty()) { System.out.println("e");
8:  } else { System.out.println("n");
9: }  }
```

The answer here is that it does compile because the code starts below the imports. Now, what about this one? Does it compile?

```
1: public class LineNumbers {
2: public void method(ArrayList list) {
3:  if (list.isEmpty()) { System.out.println("e");
4:  } else { System.out.println("n");
5: }  } }
```

For this one, you would answer "Does not compile." Since the code begins with line 1, you don't get to assume that valid imports were provided earlier. The exam will let you know what package classes are in unless they're covered in the objectives. You'll be expected to know that `ArrayList` is in `java.util`—at least you will once you get to Chapter 3 of this book!

You'll also see code that doesn't have a `main()` method. When this happens, assume the `main()` method, class definition, and all necessary imports are present. You're just being asked if the part of the code you're shown compiles when dropped into valid surrounding code.

Creating Objects

Our programs wouldn't be able to do anything useful if we didn't have the ability to create new objects. Remember that an object is an instance of a class. In the following sections, we'll look at constructors, object fields, instance initializers, and the order in which values are initialized.

Constructors

To create an instance of a class, all you have to do is write `new` before it. For example:

```
Random r = new Random();
```

First you declare the type that you'll be creating (`Random`) and give the variable a name (`r`). This gives Java a place to store a reference to the object. Then you write `new Random()` to actually create the object.

`Random()` looks like a method since it is followed by parentheses. It's called a *constructor*, which is a special type of method that creates a new object. Now it's time to define a constructor of your own:

```
public class Chick {
  public Chick() {
    System.out.println("in constructor");
  }
}
```

There are two key points to note about the constructor: the name of the constructor matches the name of the class, and there's no return type. You'll likely see a method like this on the exam:

```
public void Chick() { } // NOT A CONSTRUCTOR
```

When you see a method name beginning with a capital letter and having a return type, pay special attention to it. It is *not* a constructor since there's a return type. It's a regular method that won't be called when you write `new Chick()`.

The purpose of a constructor is to initialize fields, although you can put any code in there. Another way to initialize fields is to do so directly on the line on which they're declared. This example shows both approaches:

```
public class Chicken {
  int numEggs = 0;// initialize on line
  String name;
  public Chicken() {
    name = "Duke";// initialize in constructor
  } }
```

For most classes, you don't have to code a constructor—the compiler will supply a "do nothing" default constructor for you. There's one scenario that requires you to declare a constructor that you'll learn about in Chapter 5.

Reading and Writing Object Fields

It's possible to read and write instance variables directly from the caller. In this example, a mother swan lays eggs:

```
public class Swan {
  int numberEggs;// instance variable
  public static void main(String[] args) {
    Swan mother = new Swan();
    mother.numberEggs = 1;    // set variable
    System.out.println(mother.numberEggs);  // read variable
  }
}
```

Reading a variable is known as *getting* it. The class gets `numberEggs` directly to print it out. Writing to a variable is known as *setting* it. This class sets `numberEggs` to 1.

In Chapter 4, you'll learn how to protect the `Swan` class from having someone set a negative number of eggs.

You can even read and write fields directly on the line declaring them:

```
1: public class Name {
2:   String first = "Theodore";
3:   String last = "Moose";
4:   String full = first + last;
5: {
```

Lines 2 and 3 both write to fields. Line 4 does both. It reads the fields `first` and `last`. It then writes the field `full`.

Instance Initializer Blocks

When you learned about methods, you saw braces ({}). The code between the braces is called a *code block*. Sometimes this code is called being inside the braces. Anywhere you see braces is a code block.

Sometimes code blocks are inside a method. These are run when the method is called. Other times, code blocks appear outside a method. These are called *instance initializers*. In Chapter 5, you'll learn how to use a static initializer.

How many blocks do you see in this example? How many instance initializers do you see?

```
3: public static void main(String[] args) {
4:   { System.out.println("Feathers"); }
5: }
6: { System.out.println("Snowy"); }
```

There are three code blocks and one instance initializer. Counting code blocks is easy: you just count the number of pairs of braces. If there aren't the same number of open ({) and close (}) braces, the code doesn't compile. It doesn't matter that one set of braces is inside the `main()` method—it still counts.

When counting instance initializers, keep in mind that it does matter whether the braces are inside a method. There's only one pair of braces outside a method. Line 6 is an instance initializer.

Order of Initialization

When writing code that initializes fields in multiple places, you have to keep track of the order of initialization. We'll add some more rules to the order of initialization in Chapters 4 and 5. In the meantime, you need to remember:

- Fields and instance initializer blocks are run in the order in which they appear in the file.
- The constructor runs after all fields and instance initializer blocks have run.

Let's look at an example:

```
1: public class Chick {
2:  private String name = "Fluffy";
3:  { System.out.println("setting field"); }
4:  public Chick() {
5:    name = "Tiny";
6:    System.out.println("setting constructor");
7:  }
8:   public static void main(String[] args) {
9:     Chick chick = new Chick();
10:   System.out.println(chick.name); } }
```

Running this example prints this:

```
setting field
setting constructor
Tiny
```

Let's look at what's happening here. We start with the `main()` method because that's where Java starts execution. On line 9, we call the constructor of `Chick`. Java creates a new object. First it initializes `name` to `"Fluffy"` on line 2. Next it executes the `print` statement in the instance initializer on line 3. Once all the fields and instance initializers have run, Java returns to the constructor. Line 5 changes the value of `name` to `"Tiny"` and line 6 prints another statement. At this point, the constructor is done executing and goes back to the `print` statement on line 10.

Order matters for the fields and blocks of code. You can't refer to a variable before it has been initialized:

```
{ System.out.println(name); }  // DOES NOT COMPILE
private String name = "Fluffy";
```

You should expect to see a question about initialization on the exam. Let's try one more. What do you think this code prints out?

```
public class Egg {
  public Egg() {
    number = 5;
  }
  public static void main(String[] args) {
    Egg egg = new Egg();
    System.out.println(egg.number);
  }
  private int number = 3;
  { number = 4; } }
```

If you answered 5, you got it right. Fields and blocks are run first in order, setting *number* to 3 and then 4. Then the constructor runs, setting *number* to 5.

Distinguishing Between Object References and Primitives

Java applications contain two types of data: primitive types and reference types. In this section, we'll discuss the differences between a primitive type and a reference type.

Primitive Types

Java has eight built-in data types, referred to as the Java *primitive types*. These eight data types represent the building blocks for Java objects, because all Java objects are just a complex collection of these primitive data types. The exam assumes you are well versed in the eight primitive data types, their relative sizes, and what can be stored in them.

Table 1.1 shows the Java primitive types together with their size in bytes and the range of values that each holds.

TABLE 1.1 Java primitive types

Keyword	Type	Example
`boolean`	true or false	`true`
`byte`	8-bit integral value	`123`
`short`	16-bit integral value	`123`
`int`	32-bit integral value	`123`
`long`	64-bit integral value	`123`
`float`	32-bit floating-point value	`123.45f`
`double`	64-bit floating-point value	`123.456`
`char`	16-bit Unicode value	`'a'`

There's a lot of information in Table 1.1. Let's look at some key points:

- `float` and `double` are used for floating-point (decimal) values.
- A float requires the letter `f` following the number so Java knows it is a float.
- `byte`, `short`, `int`, and `long` are used for numbers without decimal points.
- Each numeric type uses twice as many bits as the smaller similar type. For example, `short` uses twice as many bits as `byte` does.

You won't be asked about the exact sizes of most of these types. You should know that a byte can hold a value from –128 to 127. So you aren't stuck memorizing this, let's look at how Java gets that. A byte is 8 bits. A bit has two possible values. (These are basic computer science definitions that you should memorize.) 2^8 is $2 \times 2 = 4 \times 2 = 8 \times 2 = 16 \times 2 = 32 \times 2 = 64 \times 2 = 128 \times 2 = 256$. Since 0 needs to be included in the range, Java takes it away from the positive side. Or if you don't like math, you can just memorize it.

The number of bits is used by Java when it figures out how much memory to reserve for your variable. For example, Java allocates 32 bits if you write this:

```
int num;
```

Real World Scenario

What Is the Largest int?

You do not have to know this for the exam, but the maximum number an `int` can hold is `2,147,483,647`. How do we know this? One way is to have Java tell us:

```
System.out.println(Integer.MAX_VALUE);
```

The other way is with math. An `int` is 32 bits. 2^{32} is `4,294,967,296`. Divide that by 2 and you get 2,147,483,648. Then subtract 1 as we did with bytes and you get 2,147,483,647. It's easier to just ask Java to print the value, isn't it?

There are a few more things you should know about numeric primitives. When a number is present in the code, it is called a *literal*. By default, Java assumes you are defining an `int` value with a literal. In this example, the number listed is bigger than what fits in an `int`. Remember, you aren't expected to memorize the maximum value for an `int`. The exam will include it in the question if it comes up.

```
long max = 3123456789;  // DOES NOT COMPILE
```

Java complains the number is out of range. And it is—for an `int`. However, we don't have an `int`. The solution is to add the character `L` to the number:

```
long max = 3123456789L;  // now Java knows it is a long
```

Alternatively, you could add a lowercase `l` to the number. But please use the uppercase `L`. The lowercase l looks like the number 1.

Another way to specify numbers is to change the "base." When you learned how to count, you studied the digits 0–9. This numbering system is called base 10 since there are 10 numbers. It is also known as the decimal number system. Java allows you to specify digits in several other formats:

- octal (digits 0–7), which uses the number 0 as a prefix—for example, 017
- hexadecimal (digits 0–9 and letters A–F), which uses the number 0 followed by x or X as a prefix—for example, 0xFF
- binary (digits 0–1), which uses the number 0 followed by b or B as a prefix—for example, 0b10

You won't need to convert between number systems on the exam. You'll have to recognize valid literal values that can be assigned to numbers.

Converting Back to Binary

Although you don't need to convert between number systems on the exam, we'll look at one example in case you're curious:

```
System.out.println(56);       // 56
System.out.println(0b11);     // 3
System.out.println(017);      // 15
System.out.println(0x1F);     // 31
```

First we have our normal base 10 value. We know you already know how to read that, but bear with us. The rightmost digit is 6, so it's "worth" 6. The second-to-rightmost digit is 5, so it's "worth" 50 (5 times 10.) Adding these together, we get 56.

Next we have binary, or base 2. The rightmost digit is 1 and is "worth" 1. The second-to-rightmost digit is also 1. In this case, it's "worth" 2 (1 times 2) because the base is 2. Adding these gets us 3.

Then comes octal, or base 8. The rightmost digit is 7 and is "worth" 7. The second-to-rightmost digit is 1. In this case, it's "worth" 8 (1 times 8) because the base is 8. Adding these gets us 15.

Finally, we have hexadecimal, or base 16, which is also known as hex. The rightmost "digit" is F and it's "worth" 15 (9 is "worth" 9, A is "worth" 10, B is "worth" 11, and so forth). The second-to-rightmost digit is 1. In this case, it's "worth" 16 (1 times 16) because the base is 16. Adding these gets us 31.

The last thing you need to know about numeric literals is a feature added in Java 7. You can have underscores in numbers to make them easier to read:

```
int million1 = 1000000;
int million2 = 1_000_000;
```

We'd rather be reading the latter one because the zeroes don't run together. You can add underscores anywhere except at the beginning of a literal, the end of a literal, right before a decimal point, or right after a decimal point. Let's look at a few examples:

```
double notAtStart = _1000.00;         // DOES NOT COMPILE
double notAtEnd = 1000.00_;           // DOES NOT COMPILE
double notByDecimal = 1000_.00;       // DOES NOT COMPILE
double annoyingButLegal = 1_00_0.0_0;  // this one compiles
```

Reference Types

A *reference type* refers to an object (an instance of a class). Unlike primitive types that hold their values in the memory where the variable is allocated, references do not hold the value of the object they refer to. Instead, a reference "points" to an object by storing the memory address where the object is located, a concept referred to as a *pointer*. Unlike other languages, Java does not allow you to learn what the physical memory address is. You can only use the reference to refer to the object.

Let's take a look at some examples that declare and initialize reference types. Suppose we declare a reference of type `java.util.Date` and a reference of type `String`:

```
java.util.Date today;
String greeting;
```

The *`today`* variable is a reference of type `Date` and can only point to a `Date` object. The *`greeting`* variable is a reference that can only point to a `String` object. A value is assigned to a reference in one of two ways:

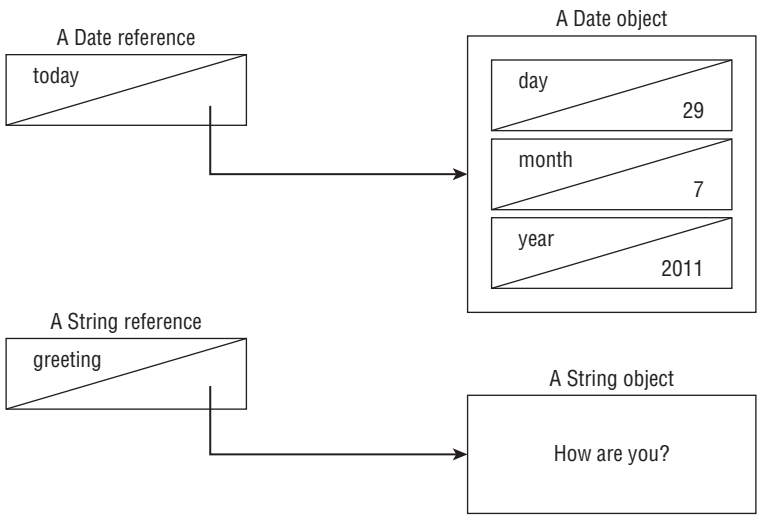
- A reference can be assigned to another object of the same type.
- A reference can be assigned to a new object using the `new` keyword.

For example, the following statements assign these references to new objects:

```
today = new java.util.Date();
greeting = "How are you?";
```

The *`today`* reference now points to a new `Date` object in memory, and *`today`* can be used to access the various fields and methods of this `Date` object. Similarly, the *`greeting`* reference points to a new `String` object, `"How are you?"`. The `String` and `Date` objects do not have names and can be accessed only via their corresponding reference. Figure 1.1 shows how the reference types appear in memory.

FIGURE 1.1 An object in memory can be accessed only via a reference.



Key Differences

There are a few important differences you should know between primitives and reference types. First, reference types can be assigned `null`, which means they do not currently refer to an object. Primitive types will give you a compiler error if you attempt to assign them `null`. In this example, *`value`* cannot point to `null` because it is of type `int`:

```
int value = null;   // DOES NOT COMPILE
String s = null;
```

Next, reference types can be used to call methods when they do not point to `null`. Primitives do not have methods declared on them. In this example, we can call a method on *`reference`* since it is of a reference type. You can tell `length` is a method because it has `()` after it. The following line is gibberish. No methods exist on *`len`* because it is an `int` primitive. Primitives do not have methods.

```
String reference = "hello";
int len = reference.length();
int bad = len.length(); // DOES NOT COMPILE
```

Finally, notice that all the primitive types have lowercase type names. All classes that come with Java begin with uppercase. You should follow this convention for classes you create as well.

Declaring and Initializing Variables

We've seen some variables already. A *variable* is a name for a piece of memory that stores data. When you declare a variable, you need to state the variable type along with giving it a name. For example, the following code declares two variables. One is named *`zooName`* and is of type `String`. The other is named *`numberAnimals`* and is of type `int`.

```
String zooName;
int numberAnimals;
```

Now that we've declared a variable, we can give it a value. This is called initializing a variable. To initialize a variable, you just type the variable name followed by an equal sign, followed by the desired value:

```
zooName = "The Best Zoo";
numberAnimals = 100;
```

Since you often want to initialize a variable right away, you can do so in the same statement as the declaration. For example, here we merge the previous declarations and initializations into more concise code:

```
String zooName = "The Best Zoo";
int numberAnimals = 100;
```

In the following sections, we'll look at how to declare multiple variables in one-line and legal identifiers.

Declaring Multiple Variables

You can also declare and initialize multiple variables in the same statement. How many variables do you think are declared and initialized in the following two lines?

```
String s1, s2;
String s3 = "yes", s4 = "no";
```

Four `String` variables were declared: *s1*, *s2*, *s3*, and *s4*. You can declare many variables in the same declaration as long as they are all of the same type. You can also initialize any or all of those values inline. In the previous example, we have two initialized variables: *s3* and *s4*. The other two variables remain declared but not yet initialized.

This is where it gets tricky. Pay attention to tricky things! The exam will attempt to trick you. Again, how many variables do you think are declared and initialized in this code?

```
int i1, i2, i3 = 0;
```

As you should expect, three variables were declared: *i1*, *i2*, and *i3*. However, only one of those values was initialized: *i3*. The other two remain declared but not yet initialized. That's the trick. Each snippet separated by a comma is a little declaration of its own. The initialization of *i3* only applies to *i3*. It doesn't have anything to do with *i1* or *i2* despite being in the same statement.

Another way the exam could try to trick you is to show you code like this line:

```
int num, String value; // DOES NOT COMPILE
```

This code doesn't compile because it tries to declare multiple variables of *different* types in the same statement. The shortcut to declare multiple variables in the same statement only works when they share a type.

To make sure you understand this, see if you can figure out which of the following are legal declarations. "Legal," "valid," and "compiles" are all synonyms in the Java exam world. We try to use all the terminology you could encounter on the exam.

```
boolean b1, b2;
String s1 = "1", s2;
double d1, double d2;
int i1; int i2;
int i3; i4;
```

The first statement is legal. It declares two variables without initializing them. The second statement is also legal. It declares two variables and initializes only one of them.

The third statement is *not* legal. Java does not allow you to declare two different types in the same statement. Wait a minute! Variables *d1* and *d2* are the same type. They are both

of type `double`. Although that's true, it still isn't allowed. If you want to declare multiple variables in the same statement, they must share the same type declaration and not repeat it. `double d1, d2;` would have been legal.

The fourth statement is legal. Although `int` does appear twice, each one is in a separate statement. A semicolon (`;`) separates statements in Java. It just so happens there are two completely different statements on the same line. The fifth statement is *not* legal. Again, we have two completely different statements on the same line. The second one is not a valid declaration because it omits the type. When you see an oddly placed semicolon on the exam, pretend the code is on separate lines and think about whether the code compiles that way. In this case, we have the following:

```
int i1;
int i2;
int i3;
i4;// DOES NOT COMPILE
```

Looking at the last line on its own, you can easily see that the declaration is invalid. And yes, the exam really does cram multiple statements onto the same line—partly to try to trick you and partly to fit more code on the screen. In the real world, please limit yourself to one declaration per statement and line. Your teammates will thank you for the readable code.

Identifiers

It probably comes as no surprise that Java has precise rules about *identifier* names. Luckily, the same rules for identifiers apply to anything you are free to name, including variables, methods, classes, and fields.

There are only three rules to remember for legal identifiers:

- The name must begin with a letter or the symbol `$` or `_`.
- Subsequent characters may also be numbers.
- You cannot use the same name as a Java *reserved word*. As you might imagine, a reserved word is a keyword that Java has reserved so that you are not allowed to use it. Remember that Java is case sensitive, so you can use versions of the keywords that only differ in case. Please don't, though.

Don't worry—you won't need to memorize the full list of reserved words. The exam will only ask you about ones you've already learned, such as `class`. The following is a list of all the reserved words in Java. `const` and `goto` aren't actually used in Java. They are reserved so that people coming from other languages don't use them by accident—and in theory, in case Java wants to use them one day.

abstract	assert	boolean	break	byte
case	catch	char	class	const*
continue	default	do	double	else
enum	extends	false	final	finally
float	for	goto*	if	implements
import	instanceof	int	interface	long
native	new	null	package	private
protected	public	return	short	static
strictfp	super	switch	synchronized	this
throw	throws	transient	true	try
void	volatile	while		

Prepare to be tested on these rules. The following examples are legal:

```
okidentifier
$OK2Identifier
_alsoOK1d3ntifi3r
__SStillOkbutKnotsonice$
```

These examples are not legal:

```
3DPointClass  // identifiers cannot begin with a number
hollywood@vine // @ is not a letter, digit, $ or _
*$coffee // * is not a letter, digit, $ or _
public   // public is a reserved word
```

Although you can do crazy things with identifier names, you shouldn't. Java has conventions so that code is readable and consistent. This consistency includes CamelCase. In CamelCase, each word begins with an uppercase letter. This makes multiple-word variable names easier to read. Which would you rather read: *`Thisismyclass name`* or *`ThisIsMyClass name`*? The exam will mostly use common conventions for identifiers, but not always. When you see a nonstandard identifier, be sure to check if it is legal. If not, you get to mark the answer "does not compile" and skip analyzing everything else in the question.

Real World Scenario

Identifiers in the Real World

Most Java developers follow these conventions for identifier names:

- Method and variables names begin with a lowercase letter followed by CamelCase.
- Class names begin with an uppercase letter followed by CamelCase. Don't start any identifiers with $. The compiler uses this symbol for some files.

Also, know that valid letters in Java are not just characters in the English alphabet. Java supports the Unicode character set, so there are more than 45,000 characters that can start a legal Java identifier. A few hundred more are non-Arabic numerals that may appear after the first character in a legal identifier. Luckily, you don't have to worry about memorizing those for the exam. If you are in a country that doesn't use the English alphabet, this is useful to know for a job.

Understanding Default Initialization of Variables

Before you can use a variable, it needs a value. Some types of variables get this value set automatically, and others require the programmer to specify it. In the following sections, we'll look at the differences between the defaults for local, instance, and class variables.

Local Variables

A *local variable* is a variable defined within a method. Local variables must be initialized before use. They do not have a default value and contain garbage data until initialized. The compiler will not let you read an uninitialized value. For example, the following code generates a compiler error:

```
4: public int notValid() {
5:  int y = 10;
6:  int x;
7:  int reply = x + y; // DOES NOT COMPILE
8:  return reply;
9: }
```

y is initialized to 10. However, because *x* is not initialized before it is used in the expression on line 7, the compiler generates the following error:

```
Test.java:5: variable x might not have been initialized
        int reply = x + y;
              ^
```

Until *x* is assigned a value, it cannot appear within an expression, and the compiler will gladly remind you of this rule. The compiler knows your code has control of what happens inside the method and can be expected to initialize values.

The compiler is smart enough to recognize variables that have been initialized after their declaration but before they are used. Here's an example:

```
public int valid() {
  int y = 10;
  int x; // x is declared here
  x = 3; // and initialized here
  int reply = x + y;
  return reply;
}
```

The compiler is also smart enough to recognize initializations that are more complex. In this example, there are two branches of code. *answer* is initialized in both of them so the compiler is perfectly happy. *onlyOneBranch* is only initialized if *check* happens to be true. The compiler knows there is the possibility for *check* to be false, resulting in uninitialized code, and gives a compiler error. You'll learn more about the `if` statement in the next chapter.

```
public void findAnswer(boolean check) {
  int answer;
  int onlyOneBranch;
  if (check) {
    onlyOneBranch = 1;
    answer = 1;
  } else {
    answer = 2;
  }
  System.out.println(answer);
  System.out.println(onlyOneBranch); // DOES NOT COMPILE
}
```

Instance and Class Variables

Variables that are not local variables are known as *instance variables* or *class variables*. Instance variables are also called fields. Class variables are shared across multiple objects.

You can tell a variable is a class variable because it has the keyword `static` before it. You'll learn about this in Chapter 4. For now, just know that a variable is a class variable if it has the `static` keyword in its declaration.

Instance and class variables do not require you to initialize them. As soon as you declare these variables, they are given a default value. You'll need to memorize everything in table 1.2 except the default value of `char`. To make this easier, remember that the compiler doesn't know what value to use and so wants the simplest type it can give the value: `null` for an object and `0`/`false` for a primitive.

TABLE 1.2 Default initialization values by type

Variable type	Default initialization value
`boolean`	`false`
`byte`, `short`, `int`, `long`	`0` (in the type's bit-length)
`float`, `double`	`0.0` (in the type's bit-length)
`char`	`'\u0000'` (NUL)
All object references (everything else)	`null`

Understanding Variable Scope

You've learned that local variables are declared within a method. How many local variables do you see in this example?

```
public void eat(int piecesOfCheese) {
  int bitesOfCheese = 1;
}
```

There are two local variables in this method. *`bitesOfCheese`* is declared inside the method. *`piecesOfCheese`* is called a method parameter. It is also local to the method. Both of these variables are said to have a *scope* local to the method. This means they cannot be used outside the method.

Local variables can never have a scope larger than the method they are defined in. However, they can have a smaller scope. Consider this example:

```
3: public void eatIfHungry(boolean hungry) {
4:  if (hungry) {
5:   int bitesOfCheese = 1;
```

```
6:  }  // bitesOfCheese goes out of scope here
7:   System.out.println(bitesOfCheese);// DOES NOT COMPILE
8: }
```

hungry has a scope of the entire method. *bitesOfCheese* has a smaller scope. It is only available for use in the `if` statement because it is declared inside of it. When you see a set of braces (`{ }`) in the code, it means you have entered a new block of code. Each block of code has its own scope. When there are multiple blocks, you match them from the inside out. In our case, the `if` statement block begins at line 4 and ends at line 6. The method's block begins at line 3 and ends at line 8.

Since *bitesOfCheese* is declared in such a block, the scope is limited to that block. When the compiler gets to line 7, it complains that it doesn't know anything about this *bitesOfCheese* thing and gives an error:

```
bitesOfCheese cannot be resolved to a variable
```

Remember that blocks can contain other blocks. These smaller contained blocks can reference variables defined in the larger scoped blocks, but not vice versa. For example:

```
16: public void eatIfHungry(boolean hungry) {
17:   if (hungry) {
18:     int bitesOfCheese = 1;
19:     {
20:        boolean teenyBit = true;
21:        System.out.println(bitesOfCheese);
22:     }
23:   }
24:   System.out.println(teenyBit);  // DOES NOT COMPILE
25: }
```

The variable defined on line 18 is in scope until the block ends on line 23. Using it in the smaller block from lines 19 to 22 is fine. The variable defined on line 20 goes out of scope on line 22. Using it on line 24 is not allowed.

The exam may attempt to trick you with questions on scope. You'll probably see a question that appears to be about something complex and fails to compile because one of the variables is out of scope. Let's try one. Don't worry if you aren't familiar with `if` statements or `while` loops yet. It doesn't matter what the code does since we are talking about scope. See if you can figure out on which line each of the five local variables goes into and out of scope:

```
11: public void eatMore(boolean hungry, int amountOfFood) {
12:  int roomInBelly = 5;
13:  if (hungry) {
14:    boolean timeToEat = true;
15:    while (amountOfFood > 0) {
16:      int amountEaten = 2;
```

```
17:       roomInBelly = roomInBelly - amountEaten;
18:       amountOfFood = amountOfFood - amountEaten;
19:     }
20:   }
21:   System.out.println(amountOfFood);
22: }
```

The first step in figuring out the scope is to identify the blocks of code. In this case, there are three blocks. You can tell this because there are three sets of braces. Starting from the innermost set, we can see where the `while` loop's block starts and ends. Repeat this as we go out for the `if` statement block and method block. Table 1.3 shows the line numbers that each block starts and ends on.

TABLE 1.3 Blocks for scope

Line	First line in block	Last line in block
`while`	15	19
`if`	13	20
Method	11	22

You'll want to practice this skill a lot. Identifying blocks needs to be second nature for the exam. The good news is that there are lots of code examples to practice on. You can look at any code example in this book on any topic and match up braces.

Now that we know where the blocks are, we can look at the scope of each variable. *`hungry`* and *`amountOfFood`* are method parameters, so they are available for the entire method. This means their scope is lines 11 to 22. *`roomInBelly`* goes into scope on line 12 because that is where it is declared. It stays in scope for the rest of the method and so goes out of scope on line 22. *`timeToEat`* goes into scope on line 14 where it is declared. It goes out of scope on line 20 where the `if` block ends. *`amountEaten`* goes into scope on line 16 where it is declared. It goes out of scope on line 19 where the `while` block ends.

All that was for local variables. Luckily the rule for instance variables is easier: they are available as soon as they are defined and last for the entire lifetime of the object itself. The rule for class (static) variables is even easier: they go into scope when declared like the other variables types. However, they stay in scope for the entire life of the program.

Let's do one more example to make sure you have a handle on this. Again, try to figure out the type of the four variables and when they go into and out of scope.

```
1:  public class Mouse {
2:    static int MAX_LENGTH = 5;
3:    int length;
```

```
4:    public void grow(int inches) {
5:      if (length < MAX_LENGTH) {
6:        int newSize = length + inches;
7:        length = newSize;
8:      }
9:     }
10: }
```

In this class, we have one class variable (*MAX_LENGTH*), one instance variable (*length*), and two local variables (*inches* and *newSize*.) *MAX_LENGTH* is a class variable because it has the `static` keyword in its declaration. *MAX_LENGTH* goes into scope on line 2 where it is declared. It stays in scope until the program ends. *length* goes into scope on line 3 where it is declared. It stays in scope as long as this `Mouse` object exists. *inches* goes into scope where it is declared on line 4. It goes out of scope at the end of the method on line 9. *newSize* goes into scope where it is declared on line 6. Since it is defined inside the `if` statement block, it goes out of scope when that block ends on line 8.

Got all that? Let's review the rules on scope:

- Local variables—in scope from declaration to end of block
- Instance variables—in scope from declaration until object garbage collected
- Class variables—in scope from declaration until program ends

Ordering Elements in a Class

Now that you've seen the most common parts of a class, let's take a look at the correct order to type them into a file. Comments can go anywhere in the code. Beyond that, you need to memorize the rules in Table 1.4.

TABLE 1.4 Elements of a class

Element	Example	Required?	Where does it go?
Package declaration	`package abc;`	No	First line in the file
Import statements	`import java.util.*;`	No	Immediately after the package
Class declaration	`public class C`	Yes	Immediately after the import
Field declarations	`int value;`	No	Anywhere inside a class
Method declarations	`void method()`	No	Anywhere inside a class

Let's look at a few examples to help you remember this. The first example contains one of each element:

```
package structure;   // package must be first non-comment
import java.util.*;  // import must come after package
public class Meerkat { // then comes the class
  double weight;       // fields and methods can go in either order
  public double getWeight() {
    return weight; }
  double height;   //  another field - they don't need to be together
 }
```

So far so good. This is a common pattern that you should be familiar with. How about this one?

```
/* header */
package structure;
// class Meerkat
public class Meerkat { }
```

Still good. We can put comments anywhere, and imports are optional. In the next example, we have a problem:

```
import java.util.*;
package structure;   // DOES NOT COMPILE
String name;  // DOES NOT COMPILE
public class Meerkat { }
```

There are two problems here. One is that the `package` and `import` statements are reversed. Though both are optional, `package` must come before `import` if present. The other issue is that a field attempts declaration outside a class. This is not allowed. Fields and methods must be within a class.

Got all that? Think of the acronym PIC (picture): package, import, and class. Fields and methods are easier to remember because they merely have to be inside of a class.

You need to know one more thing about class structure for the OCA exam: multiple classes can be defined in the same file, but only one of them is allowed to be public. The public class matches the name of the file. For example, these two classes must be in a file named `Meerkat.java`:

```
1: public class Meerkat { }
2: class Paw { }
```

A file is also allowed to have neither class be public. As long as there isn't more than one public class in a file, it is okay. On the OCP exam, you'll also need to understand inner classes, which are classes within a class.

Destroying Objects

Now that we've played with our objects, it is time to put them away. Luckily, Java automatically takes care of that for you. Java provides a garbage collector to automatically look for objects that aren't needed anymore.

All Java objects are stored in your program memory's *heap*. The heap, which is also referred to as the free store, represents a large pool of unused memory allocated to your Java application. The heap may be quite large, depending on your environment, but there is always a limit to its size. If your program keeps instantiating objects and leaving them on the heap, eventually it will run out of memory.

In the following sections, we'll look at garbage collection and the `finalize()` method.

Garbage Collection

Garbage collection refers to the process of automatically freeing memory on the heap by deleting objects that are no longer reachable in your program. There are many different algorithms for garbage collection, but you don't need to know any of them for the exam. You *do* need to know that `System.gc()` is not guaranteed to run, and you should be able to recognize when objects become eligible for garbage collection.

Let's start with the first one. Java provides a method called `System.gc()`. Now you might think from the name that this tells Java to run garbage collection. Nope! It meekly *suggests* that now might be a good time for Java to kick off a garbage collection run. Java is free to ignore the request.

The more interesting part of garbage collection is when the memory belonging to an object can be reclaimed. Java waits patiently until the code no longer needs that memory. An object will remain on the heap until it is no longer reachable. An object is no longer reachable when one of two situations occurs:

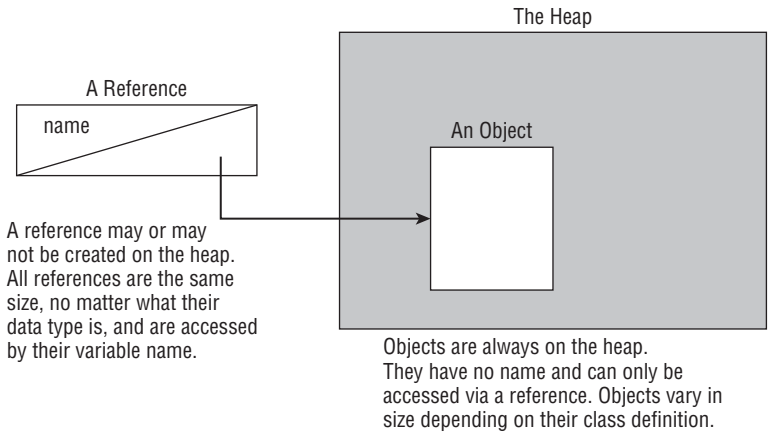
- The object no longer has any references pointing to it.
- All references to the object have gone out of scope.

Objects vs. References

Do not confuse a reference with the object that it refers to; they are two different entities. The reference is a variable that has a name and can be used to access the contents of an object. A reference can be assigned to another reference, passed to a method, or returned from a method. All references are the same size, no matter what their type is.

An object sits on the heap and does not have a name. Therefore, you have no way to access an object except through a reference. Objects come in all different shapes and sizes and consume varying amounts of memory. An object cannot be assigned to another

object, nor can an object be passed to a method or returned from a method. It is the object that gets garbage collected, not its reference.



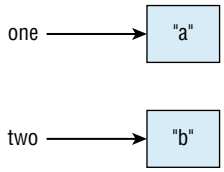
Realizing the difference between a reference and an object goes a long way toward understanding garbage collection, the `new` operator, and many other facets of the Java language. Look at this code and see if you can figure out when each object first becomes eligible for garbage collection:

```
1: public class Scope {
2:  public static void main(String[] args) {
3:    String one, two;
4:    one = new String("a");
5:    two = new String("b");
6:    one = two;
7:    String three = one;
8:    one = null;
9:  } }
```

When you get asked a question about garbage collection on the exam, we recommend you draw what's going on. There's a lot to keep track of in your head and it's easy to make a silly mistake trying to keep it all in your memory. Let's try it together now. Really. Get a pencil and paper. We'll wait.

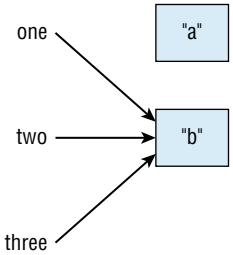
Got that paper? Okay, let's get started. On line 3, we write *one* and *two*. Just the words. No need for boxes or arrows yet since no objects have gone on the heap yet. On line 4, we have our first object. Draw a box with the string `"a"` in it and draw an arrow from the word *one* to that box. Line 5 is similar. Draw another box with the string `"b"` in it this time and an arrow from the word *two*. At this point, your work should look like Figure 1.2.

FIGURE 1.2 Your drawing after line 5



On line 6, the variable *one* changes to point to `"b"`. Either erase or cross out the arrow from *one* and draw a new arrow from *one* to `"b"`. On line 7, we have a new variable, so write the word *three* and draw an arrow from *three* to `"b"`. Notice that *three* points to what *one* is pointing to right now and not what it was pointing to at the beginning. This is why we are drawing pictures. It's easy to forget something like that. At this point, your work should look like Figure 1.3.

FIGURE 1.3 Your drawing after line 7



Finally, cross out the line between *one* and `"b"` since line 8 sets this variable to null. Now, we were trying to find out when the objects were first eligible for garbage collection. On line 6, we got rid of the only arrow pointing to `"a"`, making that object eligible for garbage collection. `"b"` has arrows pointing to it until it goes out of scope. This means `"b"` doesn't go out of scope until the end of the method on line 9.

finalize()

Java allows objects to implement a method called `finalize()` that might get called. This method gets called if the garbage collector tries to collect the object. If the garbage collector doesn't run, the method doesn't get called. If the garbage collector fails to collect the object and tries to run it again later, the method doesn't get called a second time.

In practice, this means you are highly unlikely to use it in real projects. Luckily, there isn't much to remember about `finalize()` for the exam. Just keep in mind that it might not get called and that it definitely won't be called twice.

With that said, this call produces no output when we run it:

```
public class Finalizer {
  protected void finalize()  {
```

```
    System.out.println("Calling finalize");
  }
  public static void main(String[] args) {
    Finalizer f = new Finalizer();
  } }
```

The reason is that the program exits before there is any need to run the garbage collector. While *f* is eligible for garbage collection, Java has better things to do than take out the trash constantly. For the exam, you need to know that this `finalize()` call could run zero or one time. Now for a more interesting example:

```
public class Finalizer {
  private static List objects = new ArrayList();
  protected void finalize()  {
    objects.add(this);  // Don't do this
  } }
```

Remember, `finalize()` is only run when the object is eligible for garbage collection. The problem here is that by the end of the method, the object is no longer eligible for garbage collection because a static variable is referring to it and static variables stay in scope until the program ends. Java is smart enough to realize this and aborts the attempt to throw out the object. Now suppose later in the program *objects* is set to `null`. Oh, good, we can finally remove the object from memory. Java remembers already running `finalize()` on this object and will not do so again. The lesson is that the `finalize()` call could run zero or one time. This is the exact same lesson as the simple example—that's why it's so easy to remember.

Benefits of Java

Java has some key benefits that you'll need to know for the exam:

Object Oriented Java is an object-oriented language, which means all code is defined in classes and most of those classes can be instantiated into objects. We'll discuss this more throughout the book. Many languages before Java were procedural, which meant there were routines or methods but no classes. Another common approach is functional programming. Java allows for functional programming within a class, but object oriented is still the main organization of code.

Encapsulation Java supports access modifiers to protect data from unintended access and modification. Most people consider encapsulation to be an aspect of object-oriented languages. Since the exam objectives call attention to it specifically, so do we.

Platform Independent Java is an interpreted language because it gets compiled to bytecode. A key benefit is that Java code gets compiled once rather than needing to be

recompiled for different operating systems. This is known as "write once, run everywhere." On the OCP exam, you'll learn that it is possible to write code that does not run everywhere. For example, you might refer to a file in a specific directory. If you get asked on the OCA exam, the answer is that the same class files run everywhere.

Robust One of the major advantages of Java over C++ is that it prevents memory leaks. Java manages memory on its own and does garbage collection automatically. Bad memory management in C++ is a big source of errors in programs.

Simple Java was intended to be simpler than C++. In addition to eliminating pointers, it got rid of operator overloading. In C++, you could write a + b and have it mean almost anything.

Secure Java code runs inside the JVM. This creates a sandbox that makes it hard for Java code to do evil things to the computer it is running on.

Summary

In this chapter, you saw that Java classes consist of members called fields and methods. An object is an instance of a Java class. There are three styles of comment: a single-line comment (`//`), a multiline comment (`/* */`), and a Javadoc comment (`/** */`).

Java begins program execution with a `main()` method. The most common signature for this method run from the command line is `public static void main(String[] args)`. Arguments are passed in after the class name, as in `java NameOfClass firstArgument`. Arguments are indexed starting with 0.

Java code is organized into folders called packages. To reference classes in other packages, you use an import statement. A wildcard ending an import statement means you want to import all classes in that package. It does not include packages that are inside that one. `java.lang` is a special package that does not need to be imported.

Constructors create Java objects. A constructor is a method matching the class name and omitting the return type. When an object is instantiated, fields and blocks of code are initialized first. Then the constructor is run.

Primitive types are the basic building blocks of Java types. They are assembled into reference types. Reference types can have methods and be assigned to `null`. In addition to "normal" numbers, numeric literals are allowed to begin with 0 (octal), 0x (hex), 0X (hex), 0b (binary), or 0B (binary). Numeric literals are also allowed to contain underscores as long as they are directly between two other numbers.

Declaring a variable involves stating the data type and giving the variable a name. Variables that represent fields in a class are automatically initialized to their corresponding "zero" or null value during object instantiation. Local variables must be specifically initialized. Identifiers may contain letters, numbers, $, or _. Identifiers may not begin with numbers.

Scope refers to that portion of code where a variable can be accessed. There are three kinds of variables in Java, depending on their scope: instance variables, class variables, and

local variables. Instance variables are the nonstatic fields of your class. Class variables are the static fields within a class. Local variables are declared within a method.

For some class elements, order matters within the file. The package statement comes first if present. Then comes the import statements if present. Then comes the class declaration. Fields and methods are allowed to be in any order within the class.

Garbage collection is responsible for removing objects from memory when they can never be used again. An object becomes eligible for garbage collection when there are no more references to it or its references have all gone out of scope. The `finalize()` method will run once for each object if/when it is first garbage collected.

Java code is object oriented, meaning all code is defined in classes. Access modifiers allow classes to encapsulate data. Java is platform independent, compiling to bytecode. It is robust and simple by not providing pointers or operator overloading. Finally, Java is secure because it runs inside a virtual machine.

Exam Essentials

Be able to write code using a main() method. A `main()` method is usually written as `public static void main(String[] args)`. Arguments are referenced starting with `args[0]`. Accessing an argument that wasn't passed in will cause the code to throw an exception.

Understand the effect of using packages and imports. Packages contain Java classes. Classes can be imported by class name or wildcard. Wildcards do not look at subdirectories. In the event of a conflict, class name imports take precedence.

Be able to recognize a constructor. A constructor has the same name as the class. It looks like a method without a return type.

Be able to identify legal and illegal declarations and initialization. Multiple variables can be declared and initialized in the same statement when they share a type. Local variables require an explicit initialization; others use the default value for that type. Identifiers may contain letters, numbers, $, or _. Identifiers may not begin with numbers. Numeric literals may contain underscores between two digits and begin with 1–9, 0, 0x, 0X, 0b, and 0B.

Be able to determine where variables go into and out of scope. All variables go into scope when they are declared. Local variables go out of scope when the block they are declared in ends. Instance variables go out of scope when the object is garbage collected. Class variables remain in scope as long as the program is running.

Be able to recognize misplaced statements in a class. Package and import statements are optional. If present, both go before the class declaration in that order. Fields and methods are also optional and are allowed in any order within the class declaration.

Know how to identify when an object is eligible for garbage collection. Draw a diagram to keep track of references and objects as you trace the code. When no arrows point to a box (object), it is eligible for garbage collection.

Review Questions

1. Which of the following are valid Java identifiers? (Choose all that apply)
   - **A.** `A$B`
   - **B.** `_helloWorld`
   - **C.** `true`
   - **D.** `java.lang`
   - **E.** `Public`
   - **F.** `1980_s`

2. What is the output of the following program?

```
1: public class WaterBottle {
2: private String brand;
3: private boolean empty;
4: public static void main(String[] args) {
5:   WaterBottle wb = new WaterBottle();
6:   System.out.print("Empty = " + wb.empty);
7:   System.out.print(", Brand = " + wb.brand);
8:  } }
```

   - **A.** Line 6 generates a compiler error.
   - **B.** Line 7 generates a compiler error.
   - **C.** There is no output.
   - **D.** `Empty = false, Brand = null`
   - **E.** `Empty = false, Brand =`
   - **F.** `Empty = null, Brand = null`

3. Which of the following are true? (Choose all that apply)

```
4: short numPets = 5;
5: int numGrains = 5.6;
6: String name = "Scruffy";
7: numPets.length();
8: numGrains.length();
9: name.length();
```

   - **A.** Line 4 generates a compiler error.
   - **B.** Line 5 generates a compiler error.
   - **C.** Line 6 generates a compiler error.
   - **D.** Line 7 generates a compiler error.
   - **E.** Line 8 generates a compiler error.

F. Line 9 generates a compiler error.

G. The code compiles as is.

4. Given the following class, which of the following is true? (Choose all that apply)

```
 1: public class Snake {
 2:
 3:  public void shed(boolean time) {
 4:
 5:    if (time) {
 6:
 7:    }
 8:    System.out.println(result);
 9:
10:  }
11: }
```

A. If `String result = "done";` is inserted on line 2, the code will compile.

B. If `String result = "done";` is inserted on line 4, the code will compile.

C. If `String result = "done";` is inserted on line 6, the code will compile.

D. If `String result = "done";` is inserted on line 9, the code will compile.

E. None of the above changes will make the code compile.

5. Given the following classes, which of the following can independently replace `INSERT IMPORTS HERE` to make the code compile? (Choose all that apply)

```
package aquarium;
public class Tank { }

package aquarium.jellies;
public class Jelly { }

package visitor;
INSERT IMPORTS HERE
public class AquariumVisitor {
  public void admire(Jelly jelly) { } }
```

A. `import aquarium.*;`

B. `import aquarium.*.Jelly;`

C. `import aquarium.jellies.Jelly;`

D. `import aquarium.jellies.*;`

E. `import aquarium.jellies.Jelly.*;`

F. None of these can make the code compile.

6. Given the following classes, what is the maximum number of imports that can be removed and have the code still compile?

```
package aquarium; public class Water { }

package aquarium;
import java.lang.*;
import java.lang.System;
import aquarium.Water;
import aquarium.*;
public class Tank {
  public void print(Water water) {
   System.out.println(water); } }
```

A. 0
B. 1
C. 2
D. 3
E. 4
F. Does not compile.

7. Given the following classes, which of the following snippets can be inserted in place of `INSERT IMPORTS HERE` and have the code compile? (Choose all that apply)

```
package aquarium;
public class Water {
  boolean salty = false;
}
package aquarium.jellies;
public class Water {
  boolean salty = true;
}
package employee;
 INSERT IMPORTS HERE
public class WaterFiller {
  Water water;
}
```

A.
```
import aquarium.*;
```

B.
```
import aquarium.Water;
import aquarium.jellies.*;
```

C.
```
import aquarium.*;
import aquarium.jellies.Water;
```

D.
```
import aquarium.*;
import aquarium.jellies.*;
```

E.
```
import aquarium.Water;
import aquarium.jellies.Water;
```

F. None of these imports can make the code compile.

8. Given the following class, which of the following calls print out `Blue Jay`? (Choose all that apply)

```
public class BirdDisplay {
  public static void main(String[] name) {
    System.out.println(name[1]);
} }
```

A. `java BirdDisplay Sparrow Blue Jay`

B. `java BirdDisplay Sparrow "Blue Jay"`

C. `java BirdDisplay Blue Jay Sparrow`

D. `java BirdDisplay "Blue Jay" Sparrow`

E. `java BirdDisplay.class Sparrow "Blue Jay"`

F. `java BirdDisplay.class "Blue Jay" Sparrow`

G. Does not compile.

9. Which of the following legally fill in the blank so you can run the `main()` method from the command line? (Choose all that apply)

```
public static void main(_______)
```

A. `String[] _names`

B. `String[] 123`

C. `String abc[]`

D. `String _Names[]`

E. `String... $n`

F. `String names`

G. None of the above.

10. Which of the following are legal entry point methods that can be run from the command line? (Choose all that apply)

A. `private static void main(String[] args)`

B. `public static final main(String[] args)`

C. `public void main(String[] args)`

D. `public static void test(String[] args)`

E. `public static void main(String[] args)`

F. `public static main(String[] args)`

G. None of the above.

11. Which of the following are true? (Choose all that apply)

A. An instance variable of type `double` defaults to `null`.

B. An instance variable of type `int` defaults to `null`.

C. An instance variable of type `String` defaults to `null`.

D. An instance variable of type `double` defaults to `0.0`.

E. An instance variable of type `int` defaults to `0.0`.

F. An instance variable of type `String` defaults to `0.0`.

G. None of the above.

12. Which of the following are true? (Choose all that apply)

A. A local variable of type `boolean` defaults to `null`.

B. A local variable of type `float` defaults to `0`.

C. A local variable of type `Object` defaults to `null`.

D. A local variable of type `boolean` defaults to `false`.

E. A local variable of type `boolean` defaults to `true`.

F. A local variable of type `float` defaults to `0.0`.

G. None of the above.

13. Which of the following are true? (Choose all that apply)

A. An instance variable of type `boolean` defaults to `false`.

B. An instance variable of type `boolean` defaults to `true`.

C. An instance variable of type `boolean` defaults to `null`.

D. An instance variable of type `int` defaults to `0`.

E. An instance variable of type `int` defaults to `0.0`.

F. An instance variable of type `int` defaults to `null`.

G. None of the above.

14. Given the following class in the file `/my/directory/named/A/Bird.java`:

```
INSERT CODE HERE
public class Bird { }
```

Which of the following replaces `INSERT CODE HERE` if we compile from `/my/directory`? (Choose all that apply)

A. `package my.directory.named.a;`

B. `package my.directory.named.A;`

C. `package named.a;`

D. `package named.A;`

E. `package a;`

F. `package A;`

G. Does not compile.

15. Which of the following lines of code compile? (Choose all that apply)

A. `int i1 = 1_234;`

B. `double d1 = 1_234_.0;`

C. `double d2 = 1_234._0;`

D. `double d3 = 1_234.0_;`

E. `double d4 = 1_234.0;`

F. None of the above.

16. Given the following class, which of the following lines of code can replace `INSERT CODE HERE` to make the code compile? (Choose all that apply)

```
public class Price {
      public void admission() {
                INSERT CODE HERE
              System.out.println(amount);
      } }
```

A. `int amount = 9L;`

B. `int amount = 0b101;`

C. `int amount = 0xE;`

D. `double amount = 0xE;`

E. `double amount = 1_2_.0_0;`

F. `int amount = 1_2_;`

G. None of the above.

17. Which of the following are true? (Choose all that apply)

```
public class Bunny {
       public static void main(String[] args) {
            Bunny bun = new Bunny();
} }
```

A. `Bunny` is a class.

B. `bun` is a class.

C. `main` is a class.

D. `Bunny` is a reference to an object.

E. `bun` is a reference to an object.

F. `main` is a reference to an object.

G. None of the above.

18. Which represent the order in which the following statements can be assembled into a program that will compile successfully? (Choose all that apply)

```
A: class Rabbit {}
B: import java.util.*;
C: package animals;
```

A. A, B, C

B. B, C, A

C. C, B, A

D. B, A

E. C, A

F. A, C

G. A, B

19. Suppose we have a class named `Rabbit`. Which of the following statements are true? (Choose all that apply)

```
1:  public class Rabbit {
2:   public static void main(String[] args) {
3:    Rabbit one = new Rabbit();
4:    Rabbit two = new Rabbit();
5:    Rabbit three = one;
6:    one = null;
7:    Rabbit four = one;
8:    three = null;
9:    two = null;
10:   two = new Rabbit();
11:   System.gc();
12: } }
```

A. The `Rabbit` object from line 3 is first eligible for garbage collection immediately following line 6.

B. The `Rabbit` object from line 3 is first eligible for garbage collection immediately following line 8.

C. The `Rabbit` object from line 3 is first eligible for garbage collection immediately following line 12.

D. The `Rabbit` object from line 4 is first eligible for garbage collection immediately following line 9.

E. The `Rabbit` object from line 4 is first eligible for garbage collection immediately following line 11.

F. The `Rabbit` object from line 4 is first eligible for garbage collection immediately following line 12.

20. What is true about the following code? (Choose all that apply)

```
public class Bear {
  protected void finalize() {
    System.out.println("Roar!");
}
```

```
public static void main(String[] args) {
  Bear bear = new Bear();
  bear = null;
  System.gc();
} }
```

A. `finalize()` is guaranteed to be called.

B. `finalize()` might or might not be called

C. `finalize()` is guaranteed not to be called.

D. Garbage collection is guaranteed to run.

E. Garbage collection might or might not run.

F. Garbage collection is guaranteed not to run.

G. The code does not compile.

21. What does the following code output?

```
1: public class Salmon {
2:  int count;
3:  public void Salmon() {
4:    count = 4;
5:  }
6: public static void main(String[] args) {
7:  Salmon s = new Salmon();
8:  System.out.println(s.count);
9: } }
```

A. 0

B. 4

C. Compilation fails on line 3.

D. Compilation fails on line 4.

E. Compilation fails on line 7.

F. Compilation fails on line 8.

22. Which of the following are true statements? (Choose all that apply)

A. Java allows operator overloading.

B. Java code compiled on Windows can run on Linux.

C. Java has pointers to specific locations in memory.

D. Java is a procedural language.

E. Java is an object-oriented language.

F. Java is a functional programming language.

23. Which of the following are true? (Choose all that apply)

A. `javac` compiles a `.class` file into a `.java` file.

B. `javac` compiles a `.java` file into a `.bytecode` file.

C. `javac` compiles a `.java` file into a `.class` file.

D. Java takes the name of the class as a parameter.

E. Java takes the name of the `.bytecode` file as a parameter.

F. Java takes the name of the `.class` file as a parameter.